Hybrid-JIT : Compilateur JIT Matériel/Logiciel pour les Processeurs VLIW Embarqués

Simon Rokicki, Erven Rohou, Steven Derrien

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Hybrid-JIT : Compilateur JIT Matériel/Logiciel pour les Processeurs VLIW Embarqués

Simon Rokicki, Erven Rohou, Steven Derrien

Université de Rennes 1

Résumé
La compilation JIT (*Just-In-Time*) est largement utilisée dans les systèmes embarqués actuels (principalement grâce à la machine virtuelle Java). Lorsque l'architecture ciblée est un processeur VLIW (*Very Long Intruction Word*), la compilation JIT devient plus complexe à cause de l'étape d'ordonnancement des instructions et d'allocation des registres, réduisant ainsi ses bénéfices. Dans ce travail, nous présentons un compilateur JIT hybride dans lequel la gestion du JIT est réalisée de manière logicielle tandis qu'un accélérateur matériel est utilisé pour les phases plus coûteuses telles que l'ordonnancement des instructions. Une étude expérimentale montre que cette approche mène à une compilation jusqu'à 15 fois plus rapide et 18 fois moins coûteuse en énergie par rapport à une approche purement logicielle.

1. Introduction

La compilation JIT (*Just-in-Time*) est apparue dans les années 60 [3], puis s'est démocratisée dans le milieu des années 90 quand un compilateur JIT a été ajouté à la Machine Virtuelle Java en plus de son interpréteur. Le principe d'un compilateur JIT est le même que celui d'un compilateur statique : il génère du code natif à partir d'une représentation intermédiaire de plus haut niveau, typiquement un *bytecode*. La principale différence est qu'il opère au dernier moment : la compilation d'une fonction a lieu la première fois qu'elle est appelée. L'intérêt principal de cette méthode est de permettre au compilateur d'utiliser des paramètres spécifiques à l'exécution en cours. En contrepartie, le temps de compilation devient une part du temps d'exécution. Il faut donc que le temps dépensé à optimiser une portion de code soit compensé par le temps d'exécution gagné. Cette contrainte limite les optimisations qui peuvent être réalisées à des algorithmes rapides, de complexité linéaire.
Dans un système embarqué, la compilation JIT présente un défi encore plus grand. Premièrement parce que ces systèmes sont souvent basés sur des architectures statiquement ordonnancées (par exemple un processeur VLIW). Ces architectures sont très dépendantes de la qualité du code généré et plus particulièrement de la qualité de l'ordonnancement des instructions et de l'allocation des registres. De plus, les fortes contraintes imposées aux systèmes embarqués limitent d'autant plus le compilateur JIT.
Nous proposons ici une solution pour réduire le coût de la compilation JIT pour les systèmes multi-coeur hétérogènes basés sur les VLIW[2, 10]. Notre approche repose sur une chaîne de compilation JIT matérielle/logicielle : les optimisations basiques sont résolues de manière logicielle tandis qu'un accélérateur matériel est utilisé pour les tâches les plus coûteuses (ordonnancement des instructions et allocation des registres, reconnus comme étant la partie critique quand on cible un VLIW).

Notre travail se base sur des algorithmes classiques d'ordonnancement. Nous avons défini et implémenté un *back-end* de compilation accéléré grâce à du matériel spécialisé. A notre connaissance, ce travail est le seul à proposer un tel flot de compilation complet et testé sur FPGA. **Nous voyons ce résultat comme une première étape démontrant la faisabilité d'un compilateur JIT matériel rapide et peu coûteux.** Plus précisément, nos contributions sont les suivantes :

— La définition d'une plateforme logicielle/matérielle permettant une compilation dynamique peu coûteuse sur des architectures multi-coeurs hétérogènes basées sur des VLIW,
— La mise au point d'un accélérateur matériel pour résoudre le problème d'ordonnancement des instructions et d'allocation des registres, ainsi que son implémentation et son test sur FPGA,
— Une estimation des coûts en surface, temps et énergie pour la technologie ASIC 65 nm qui démontrent un gain à la fois en énergie et en temps de compilation.

Le document est organisé de la manière suivante. La Section 2 décrit en détail notre approche, évaluée de façon quantitative et qualitative en Section 3. La Section 4 fournie une comparaison détaillée avec les précédents travaux sur un domaine similaire. Enfin, la Section 5 conclut ce document et offre un aperçu des futures directions de recherches possibles.

2. Approche proposée

Dans cette partie, nous détaillons notre approche pour accélérer la compilation JIT. En particulier, nous décrivons un prototype matériel original, appelé Hybrid-JIT. La plateforme Hybrid-JIT est une chaîne de compilation logicielle/matérielle ayant pour rôle de générer dynamiquement du code pour VLIW. Elle est illustrée Figure 1 dans le contexte d'une architecture multi-coeurs hétérogène basée sur des VLIW. Les processeurs VLIW (③), qui sont utilisés pour exécuter les applications, ont tous une configuration différente (nombre de voies, taille de la file de registre, profondeur de pipeline).
Le processeur JIT (①) est un simple processeur à une voie en charge de transformer le *bytecode* en un binaire spécifique à une configuration du VLIW. Il est accompagné d'un accélérateur matériel (②) permettant d'ordonnancer rapidement les instructions d'une procédure.

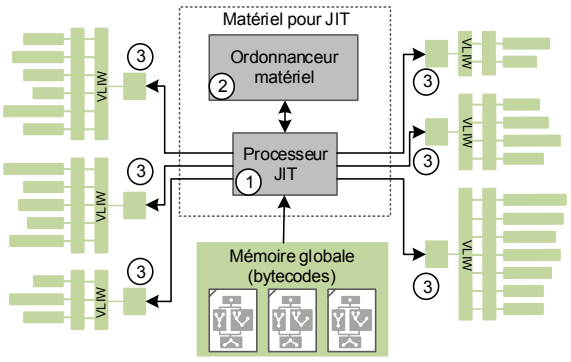

FIGURE 1 – Idée de la plateform Hybrid-JIT

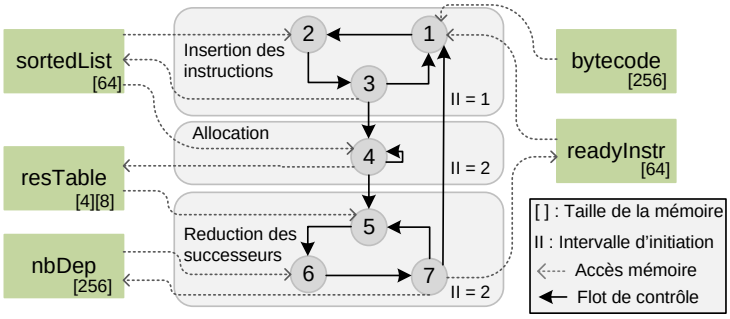

FIGURE 2 – Organisation de l'ordonnanceur

Les sous-sections traiteront du fonctionnement global du système, du choix du *bytecode* et du fonctionnement de l'accélérateur matériel.

2.1. Aperçue de la plateforme Hybrid-JIT

Comme expliqué précédemment, nous utilisons un processeur dédié à la compilation JIT dans notre système. Cela permet principalement de pouvoir compiler une fonction pendant l'exécution d'une autre tâche sur le VLIW. Le surcoût de la compilation JIT est ainsi réduit. A chaque fois qu'une fonction est appelée, le processeur suit le schéma suivant :

1. Lecture du *bytecode* dans la mémoire principale. Les instructions de chaque bloc sont ordonnancées sur les unités d'exécution du VLIW ciblé par l'accélérateur matériel. Chaque bloc est traité selon son ordre d'apparition dans la procédure.
2. Une fois tous les blocs traités (et une fois que leur taille finale est connue), le compilateur JIT va résoudre les branchements en ajoutant l'adresse correcte.
3. La dernière étape est l'écriture des binaires générés dans la mémoire d'instructions du VLIW, en effaçant si besoin une ancienne version de la procédure.

Ainsi, la gestion de la compilation JIT est réalisée de manière logicielle sur le processeur JIT et les étapes les plus coûteuses sont accélérées avec du matériel spécialisé. L'utilisation de cet accélérateur matériel réduit grandement le temps de compilation dynamique tout en ayant un faible impact sur la flexibilité du JIT puisqu'il est toujours possible d'ajouter une composante logicielle.
Un des objectifs de Hybrid-JIT est de réduire le coût en énergie d'une compilation. Dans ce contexte, augmenter la consommation globale du système en ajoutant un coeur peut paraître contre-productif. Cependant, puisque le compilateur JIT n'est utilisé que sporadiquement, le processeur et son accélérateur matériel peuvent être désactivés entre deux utilisations en utilisant le *power gating*. Ainsi, la partie de la puce dédiée au JIT consomme de l'énergie uniquement quand c'est nécessaire. De plus, l'utilisation d'un circuit spécialisé réduit le coût en énergie d'une compilation par rapport à une approche logicielle (voir Section 3). Ce coeur augmente également le budget silicium, mais dans le contexte du *Dark Silicon* [9], sacrifier une portion du budget en transistor pour améliorer l'efficacité énergétique du système est parfaitement acceptable.

2.2. Présentation du bytecode

Nous avons fait le choix d'utiliser un *bytecode* personnalisé permettant de faciliter la phase d'ordonnancement des instructions et d'allocation des registres. Un exemple détaillé permettant de mieux comprendre notre *bytecode* est donné Figure 3. Comme on peut le voir, chaque instruction est accompagnée du nombre de prédécesseurs et de successeurs dans le graphe de flot de données, de sa priorité et de la liste des instructions dépendant d'elle. Grâce à ce format, l'accélérateur matériel aura un accès rapide à toutes ces informations.

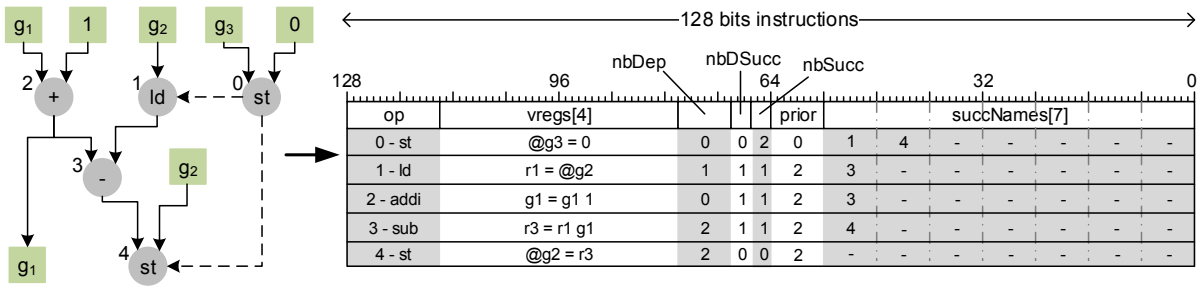

FIGURE 3 – Exemple de flot de données et du bytecode correspondant.

2.3. Accélérateur matériel pour l'ordonnancement d'instructions

Comme nous l'avons expliqué précédemment, le coeur de notre contribution est l'accélérateur matériel pour résoudre le problème de l'ordonnancement des instructions et l'allocation des registres. Nous avons utilisé un outil de synthèse de haut-niveau (HLS : *High-Level Synthesis*) pour concevoir ce circuit.
L'accélérateur est basé sur l'algorithme de *list scheduling* pour résoudre l'ordonnancement des instructions. L'allocation des registres est réalisée durant la phase d'ordonnancement : les registres sont alloués de façon gloutonne aux variables locales. Cette méthode est celle utilisée par Agosta et al. dans leur compilateur JIST [1].
L'accélérateur matériel, qui est illustré Figure 2, fonctionne de la manière suivante

1. L'insertion des instructions prêtes dans la file de priorité. Cette étape prend une instruction pour laquelle toutes les dépendances sont résolues dans `readyInstr` (étape ①) et l'insère dans la liste triée `sortedList` en fonction de sa priorité (étapes ② et ③) ;

2. L'ordonnancement d'une instruction : pour chaque unité d'exécution disponible, l'algorithme va trouver l'instruction compatible la plus prioritaire (en tête de la liste triée) et la placer dans la table de réservation `resTable` (étape ④). C'est également à cette étape que l'allocation gloutonne d'un registre est faite pour stocker le résultat ;

3. La phase de réduction des dépendances : pour chaque successeur de l'instruction présente dans la table de réservation (étape ⑤), l'algorithme décrémente le nombre de dépendances restantes (stocké dans `nbDep`) (étape ⑥). Si l'un de ces successeurs atteint zéro dépendance, il pourra être inséré dans `readyInstr` pour la prochaine exécution de l'étape 1 (étape ⑦).

Ces trois opérations sont répétées jusqu'à ce que toutes les instructions soient ordonnancées.

3. Etude expérimentale

Cette partie résume notre étude expérimentale. La première sous partie présente le protocole expérimental. La suivante présente et commente les différents résultats.

3.1. Protocole expérimental

La partie statique de notre flot de compilation est basée sur un compilateur ciblant le ST200. Notre but est d'être capable de bénéficier de toutes les optimisations pour un VLIW. Nous avons ainsi généré du code assembleur pour un ST231 que nous avons ensuite utilisé pour générer notre *bytecode*.
Pour démontrer la faisabilité de notre approche, nous avons développé et validé une version sur FPGA utilisant un unique VLIW. Ce prototype a été testé sur une carte Altera Cyclone IV. Le processeur JIT utilisé était un Nios II. L'accélérateur matériel, généré en utilisant l'outil de HLS Catapult, est accessible en tant que instruction personnalisée du Nios.
Une version purement logicielle de notre compilateur JIT est implémentée sur le Nios pour servir de référence. Cette version utilise exactement le même algorithme et le même *bytecode* que Hybrid-JIT. Le binaire pour le JIT logiciel (compilé en -O3) représente 24 kB, comparé à 7 kB pour la partie logicielle de Hybrid-JIT. Nous avons évalué notre système sur 11 benchmarks provenant de la suite Mediabench.
Le système a également été compilé vers du ASIC 65 nm en utilisant Synopsys DC Compiler. La fréquence maximale obtenue est 250 MHz. La surface utilisée est équivalente à celle d'un VLIW à 4 voies.

3.2. Résultats

Les résultats expérimentaux sont organisés en deux parties : nous comparons notre système à sa version logicielle d'abord sur le temps de compilation puis sur le coût énergétique de cette compilation.

3.2.1. Temps de compilation

La première expérience permet de mesurer l'accélération apportée par l'utilisation de notre plateforme. Nous avons utilisé des moniteurs matériels pour avoir une estimation précise au cycle prêt du temps d'exécution d'un programme. Nous avons ensuite réalisé la compilation des différents benchmarks en utilisant dans un premier cas Hybrid-JIT puis la version logicielle, exécutée sur le Nios. Les résultats de cette expérience sont présentés sur la Figure 4 (première barre) et montrent une accélération moyenne de 13x, allant jusqu'à 17x dans le meilleur cas.

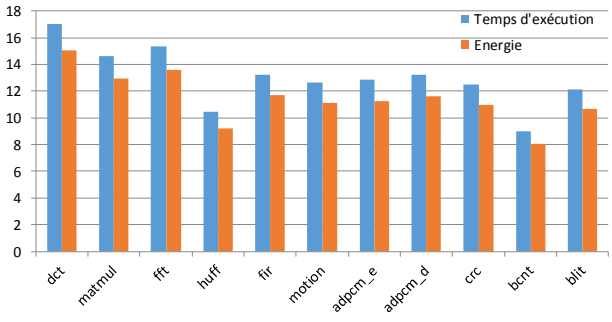

FIGURE 4 – Gain lors d'une compilation avec Hybrid-JIT comparé à une approche logicielle.

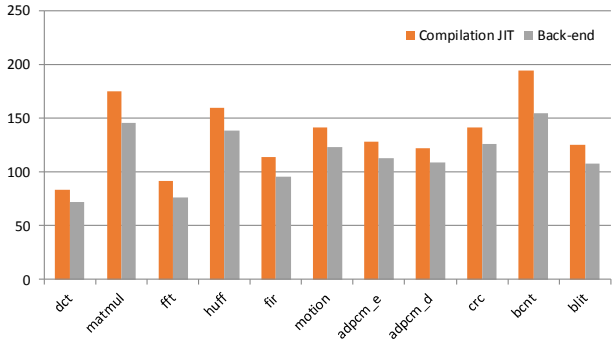

FIGURE 5 – Nombre de cycle pour compiler une instruction.

De cette expérience, nous avons également dérivé une mesure d'efficacité de notre compilateur JIT : le nombre de cycles nécessaires pour compiler une instruction du *bytecode*. Ces résultats sont représentés sur la Figure 5 pour la compilation complète mais aussi pour le *back-end* uniquement (c'est à dire pour l'ordonnancement des instructions, l'allocation de registres et la génération de l'instruction). Cette expérience montre qu'il faut en moyenne 134 cycles pour compiler une instruction.

3.2.2. Coût énergétique de la compilation

La deuxième expérience permet de mesurer le gain en énergie dû à l'utilisation de notre accélérateur matériel. Nous avons réalisé une estimation de la consommation en énergie de notre plateforme. Cette estimation est basée sur une simulation post-synthèse afin d'utiliser des données de transition des transistors réalistes.
Pour chaque benchmark, nous avons comparé le coût en énergie d'une compilation avec et sans l'accélérateur matériel. Les résultats sont représentés sur la deuxième barre de la Figure 4. On peut observer que l'utilisation d'Hybrid-JIT réduit le coût de la compilation de 8 à 15 fois.

4. État de l'art

Il n'existe que peu de travaux traitant de la compilation JIT pour les architectures statiquement ordonnancées comme les VLIW ou les CGRA [1, 6, 8]. Agosta et al. et Dupont de Dinechin s'intéressent au problème d'ordonnancement des instructions qu'ils identifient comme étant la phase critique. Ils proposent deux méthodes pour résoudre ce problème, basées sur des algorithmes gloutons : le *scoreboard scheduling* [6] et le *list scheduling* [1]. Dans les deux cas, la compilation JIT est faite de manière logicielle et est exécutée sur le VLIW.

Le travail d'Agosta et al. ne rapporte aucun chiffre sur le temps de compilation ce qui empêche toute comparaison. Cependant, l'algorithme choisi étant un *list scheduling*, on peut espérer des résultats similaires à ceux obtenus avec l'implémentation logicielle de la section 3. Avec les résultats donnés par Dupont de Dinechin, nous avons pu estimer qu'il lui fallait environ 430 cycles pour ordonnancer une instruction. Notre compilateur JIT accéléré est capable de compiler une instruction en 134 cycles en moyenne. Cette compilation comprend non seulement l'ordonnancement mais également l'allocation de registres et la génération de l'instruction. Nous sommes donc plus de 4 fois plus rapides que son outil.
Les travaux de Carbon et al. [5] sont également très similaires aux nôtres. Ils proposent d'utiliser du matériel spécialisé pour accélérer la compilation JIT avec LLVM. Pour ceci, ils ont implémenté une interface matérielle pour manipuler les arbres rouges et noirs qui peuvent être utilisés dans de nombreuses phases de la génération de code. Ils arrivent ainsi à une réduction de 15% du temps de compilation. L'idée de l'approche est similaire mais appliquée à la compilation JIT pour processeur superscalaire à exécution dans le désordre, où il n'y a pas de goulot d'étranglement aussi marqué que pour les VLIW.
La traduction de binaires a beaucoup été utilisée dans le contexte des architectures statiquement ordonnancées. Un exemple bien connu est le CMS (*Code Morphing Software*) de Transmeta [7] : le système était capable d'exécuter du code x86 sur architecture VLIW grâce à une étape de traduction binaire. Cette approche a souvent été critiquée à cause de son coût lors d'une première exécution de code généraliste. CMS était basé sur un procédé d'optimisation en plusieurs étapes qui commence par une interprétation du code puis détecte les points chauds pour y appliquer des optimisations plus agressives. Ces optimisations sont toujours suivies par une étape d'ordonnancement des instructions pour générer du code VLIW. De même, l'architecture Denver de Nvidia [4] propose une architecture *in-order* à 7 voies. Deux modes d'exécution sont possibles : dans un premier temps, le décodeur d'instructions peut décoder jusqu'à deux instructions par cycle. Ce mode permet d'utiliser 2 des 7 unités d'exécution disponibles. Les instructions ainsi décodées sont stockées et pourront être réutilisées sans avoir recours au décodeur d'instructions. Il sera alors possible d'utiliser plus d'unités d'exécution. De plus, un autre processus est en charge de lire ces instructions décodées et d'optimiser le code pour plus de performances dans les exécutions futures. Comme l'architecture exécute les instructions dans l'ordre, une étape d'ordonnancement des instructions est primordiale après une optimisation du code.
Les approches de Transmeta et de Nvidia utilisent toutes deux une étape d'ordonnancement des instructions faite de manière logicielle. Chacune des deux approches pourrait également profiter d'une réduction du coût de cet ordonnancement en utilisant du matériel spécialisé.

5. Conclusion

Dans ce document, nous avons proposé une approche permettant une compilation JIT rapide et peu coûteuse en énergie pour les architectures VLIW. Cette approche est basée sur un accélérateur matériel qui permet de résoudre le goulot d'étranglement d'un compilateur pour VLIW : l'ordonnancement des instructions et l'allocation des registres. Nous avons montré que notre approche permet de diviser le temps de compilation par 13 en moyenne et le coût en énergie par 11. Nous voyons ce résultat comme une première étape vers un compilateur JIT efficace en énergie et très rapide, permettant une migration des tâches dans un système hétérogène. Notre prochaine étape est l'extension de notre méthode à des architectures CGRA (*Coarse Grain Reconfigurable Architecture*), offrant plus de possibilités et de meilleurs compromis performance énergie mais également plus complexes pour un compilateur.

Bibliographie

1. Agosta (G.), Reghizzi (S. C.), Falauto (G.) et Sykora (M.). – JIST : Just-In-Time scheduling translation for parallel processors. *Scientific Programming*, vol. 13, n3, 2005, pp. 239–253.
2. Anjam (F.), Kong (Q.), Seedorf (R.) et Wong (S.). – A run-time task migration scheme for an adjustable issue-slots multi-core processor. – In *Proceedings of the 8th International Conference on Reconfigurable Computing : Architectures, Tools and Applications*, *ARC'12*, ARC'12, pp. 102–113, Berlin, Heidelberg, 2012. Springer-Verlag.
3. Aycock (J.). – A Brief History of Just-In-Time. *ACM Comput. Surv.*, vol. 35, n2, juin 2003, pp. 97–113.
4. Boggs (D.), Brown (G.), Tuck (N.) et Venkatraman (K.). – Denver : NVIDIA's First 64-bit ARM Processor. *Micro*, 2015.
5. Carbon (A.), Lhuillier (Y.) et Charles (H.-P.). – Hardware Acceleration of Red-Black Tree Management and Application to Just-In-Time Compilation. *Journal of Signal Processing Systems*, 2014, pp. 1–21.
6. De Dinechin (B. D.). – Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors. *In : Euro-Par 2008–Parallel Processing*, pp. 370–381. – Springer, 2008.
7. Dehnert (J. C.), Grant (B. K.), Banning (J. P.), Johnson (R.), Kistler (T.), Klaiber (A.) et Mattson (J.). – The Transmeta Code Morphing^{TM} Software : Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. – In *CGO*, pp. 15–24. IEEE Computer Society, 2003.
8. Ferreira (R.), Duarte (V.), Meireles (W.), Pereira (M.), Carro (L.) et Wong (S.). – A Just-In-Time Modulo Scheduling for Virtual Coarse-Grained Reconfigurable Architectures. – In *SAMOS XIII*, pp. 188–195. IEEE, 2013.
9. Jimborean (A.), Koukos (K.), Spiliopoulos (V.), Black-Schaffer (D.) et Kaxiras (S.). – Fix the Code. Don't Tweak the Hardware : A New Compiler Approach to Voltage-Frequency Scaling. – In *CGO '14*, New York, NY, USA, 2014. ACM.
10. Wong (S.), Anjam (F.) et Nadeem (F.). – Dynamically Reconfigurable Register File for a Softcore VLIW Processor. – In *DATE*. European Design and Automation Association, 2010.